Lecture Notes for

Managing and Mining Multiplayer Online Games

Summer Term 2019

Chapter 9: Artificial Intelligence

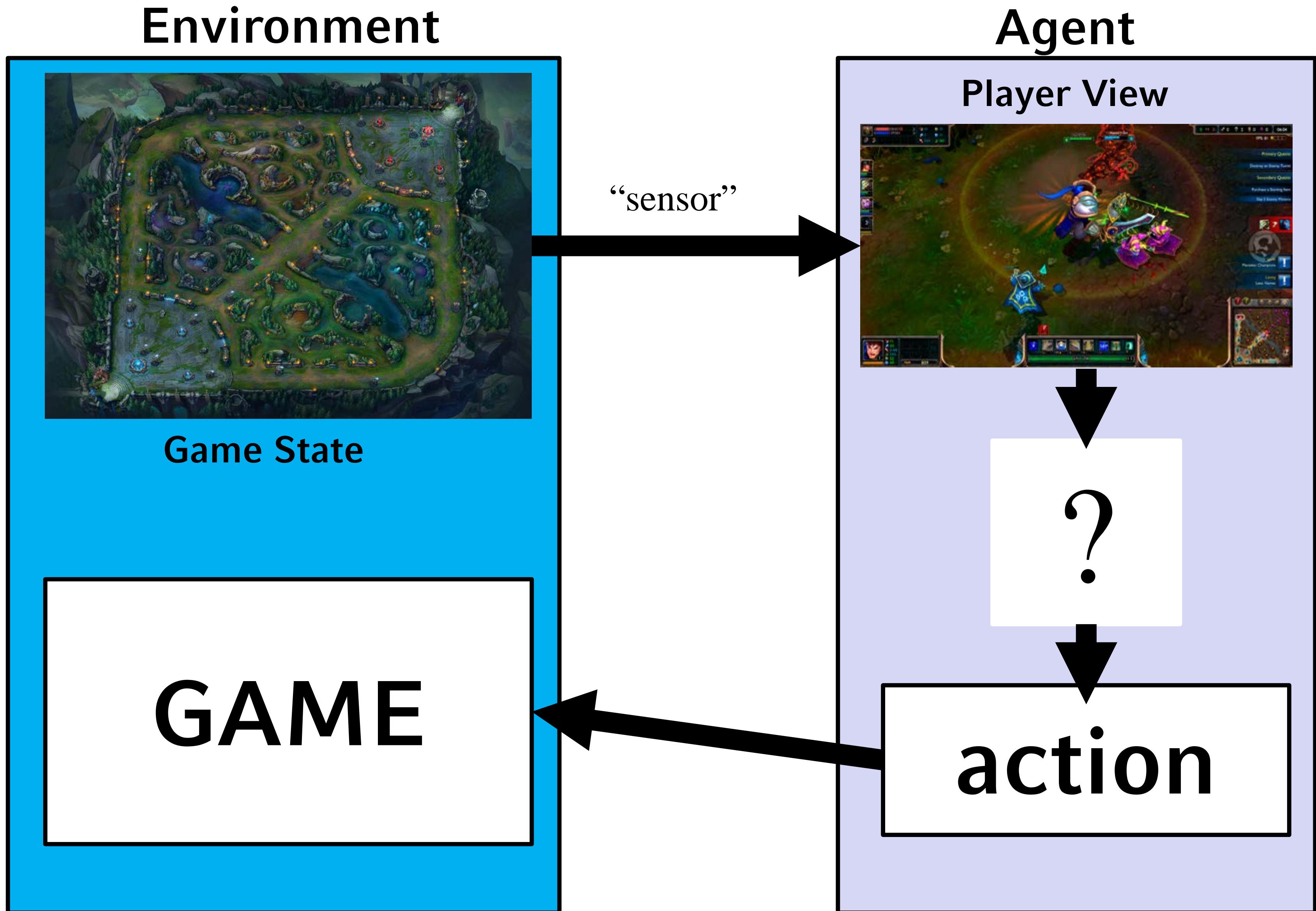
Lecture Notes © 2012 Matthias Schubert

http://www.dbs.ifi.lmu.de/cms/VO_Managing_Massive_Multiplayer_Online_Games

Chapter Overview

- What is Artificial Intelligence?
- Environments, Agents, Actions Rewards
- Sequential Decision Making
  - Classical Search
  - Planning with Uncertainty
  - Model-free Reinforcement Learning
  - Monte-Carlo and Temporal Difference Learning
  - Q-Learning
- Adversarial Search
  - Minimax
  - Alpha-Beta Pruning
  - Monte Carlo Tree Search

What is Artificial Intelligence?



Environment

Represents the world in which the agent is acting.

(e.g. a game, a simulation or a robot)

- provides information about the state (e.g. view of Game State)
- receives action and reacts to the them

Properties of Environments

- partially / fully observable
- with known model/ model free
- deterministic / non-deterministic
- single vs. multi-agent
- competitive vs. collaborative
- static / dynamic / semi-dynamic
- discrete / continuous (states and/or actions)

Agents

Autonomous entity within the environment.

types of agents:

- simple reflex agent
  - condition-action-rule
    (example: If car-in-front-is-braking then initiate-braking.)
- model-based reflex agents (add internal state from history)
- goal-based agents (works towards a goal)
- **utility-based agents** (optimizes rewards/minimizes costs)
- **learning agents** (learns how to optimize rewards/costs)

Example: „Autocamp 2000“ (simple reflex agent)

Example for a Bots: (http://www.gamespy.com/articles/489/489833p1.html)

1) If invited by any group => join group
2) If in a group => follow behind the leader
3) If sees a monster => attack
4) If someone says something ending in a question mark
   => respond by saying "Dude?"
5) If someone says something ending in an exclamation point
   => respond by saying "Dude!"
6) If someone says something ending with a period
   => respond by randomly saying one of three things: "Okie“, "Sure“, or "Right on"
7) EXCEPTION: If someone says something directly to you by mentioning your name, respond by saying "Lag."

Example

KillSwitch: [Shouting] Does anyone want to join our hunting party?

Farglik: [Powered by the Autocamp 2000] Dude?

[KillSwitch invites Farglik to join the group.]

[Farglik joins the group]

KillSwitch: We're gonna go hunt wrixes.

Farglik: Right on.

[The group of players runs out, Farglik following close behind. Farglik shoots at every little monster they pass.]

KillSwitch: Why are you attacking the durneys?

Farglik: Dude?

KillSwitch: The durneys, the little bunny things -- why do you keep shooting at them?

Farglik: Dude?

KillSwitch: Knock it off guys, I see some wrixes up ahead. Let's do this.

Farglik: Right on.

[The group encounters a bunch of dangerous wrixes, but they gang up and shoot every one of them.]

KillSwitch: We rock!

Farglik: Dude!

Troobacca: We so OWNED them!

Farglik: Dude!

Example

KillSwitch: Uh oh, hang on. Up ahead are some Sharnaff bulls. We can't handle them, so don't shoot.

Farglik: Okie.

[Farglik shoots one of the Sharnaff bulls.]

[The bull attacks; Trobacca and several other party members are killed before they beat it.]

KillSwitch: You IDIOT! Farglik why did you shoot at them?

Farglik: Lag.

KillSwitch: Well don't do it again.

Farglik: Sure.

[Farglik shoots at another Sharnaff bull.]

[The entire party is slaughtered except for Farglik.]

[... Farglik stands there, alone, for several hours ...]

Planet Fargo- The Automated Online Role-Player
By Dave Kosak
http://www.gamespy.com/articles/489/489833p1.html

Sequential Decision Making

- behavior is a sequence of actions
- sometimes immediate rewards must be sacrificed to acquire rewards in the future

  example: spend gold to build a gold mine
- short-rewards might be very unlikely in a situation
  example: score a goal from your own half in football

=> intelligent behavior needs to plan ahead

Deterministic Sequential Planning

- Set of states $\boldsymbol{S} = \{s_1,..,s_n\}$
- Set of actions ***A(s)*** for each state $s \in S$
- Reward function **R: R(s)** (if negative = cost function)
- Transition function **T: S×A => S: t(s,a) = s'**

  (deterministic case !!)
- this implies:
  - episode = $s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, s_3 \ldots, s_l, a_l, r_l, s_{l+1}$
  - *reward of the episode:* $\sum_{i=1}^{l} \gamma^i r_i$ *with* $0 < \gamma \leq 1$

    (γ=1: all rewards count the same, γ <1: early rewards count more)
  - ***sometimes****: process terminates when reaching a terminal state* s^T *(Game Over !!!) or process end after k moves.*

Deterministic Sequential Planning

- Static, discrete, deterministic, known and fully observable environments:
  - S, A are discrete sets and known
  - t(s,a) has a deterministic state s'
  - Agent knows the current state

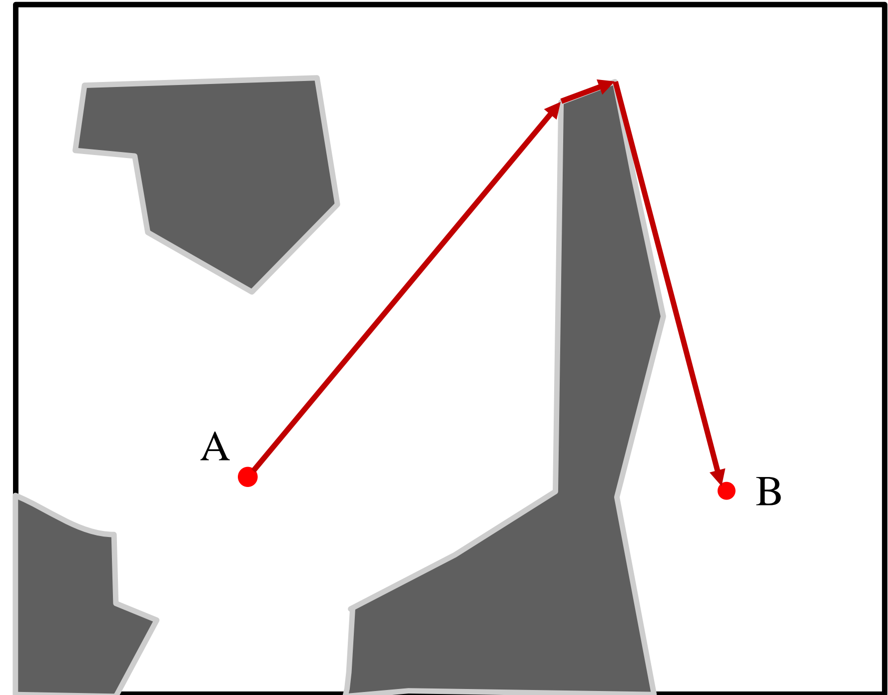
- **goal**: find a sequence of actions (path) that maximize the cumulated future rewards.

examples:

- routing from A to B on the map or find an exit
- riddles like the goat, wolf, cabbage transport problem

Routing in Open Environments

- open environment: 2D Space (IR^2)
- agents can move freely
- obstacles block direct connections
- presenting obstacles with:
  - ***polygons***
  - pixel-presentation
  - any geometric form (Circle, Ellipse, …)

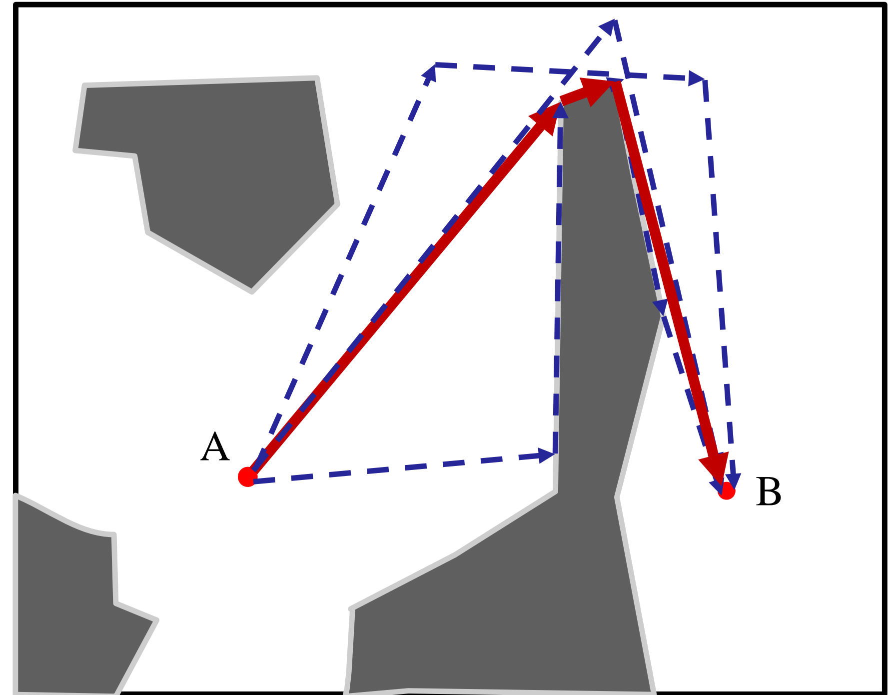


solution for polygon presentation:

- deriving a graph for the map containing the shortest routes (visibility graph)
- integrating start and goal
- use of pathfinding algorithms like Dijkstra or A*

Visibility Graph

- finding the shortest path in an open environment is a search over an infinite search area
- **solution**: restrict the search area with the following properties of optimal paths:
  - **waypoints of every shortest path are either start, goal or corners of obstacle-polygons.**
  - **paths cannot intersect polygons.**



- The shortest path in the open environment U is also part of the visibility graph $G_U(V,E)$.

Visibility Graph

Environment: *U*

- Set of polygons $U=(P_1, \ldots, P_n)$ **(Obstacles)**
- Polygon P: planar cyclic graph: $P = (V_P, E_P)$

Visibility graph: $G_U(V,E)$

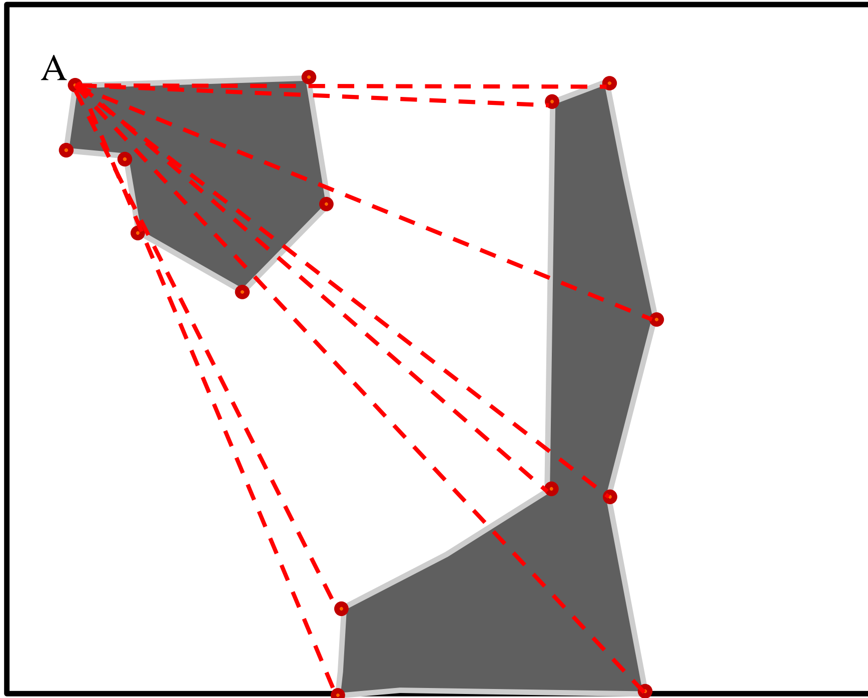
- *Nodes:* Corners of polygons $P = \{V_1, \ldots, V_l)$ in *U:* $V_U = \bigcup_{P \in U} V_P$
- *Edges:* All edges of polygons with all edges of nodes from different polygons that do not intersect another polygon-edge.

$$E_U = \bigcup_{P \in U} E_P \cup \{(x, y) | x \in P_i \wedge y \in P_j \wedge i \neq j \wedge \underset{P \in U}{\forall} \underset{e \in E_P}{\forall} : (x, y) \cap e = \{\}\}$$

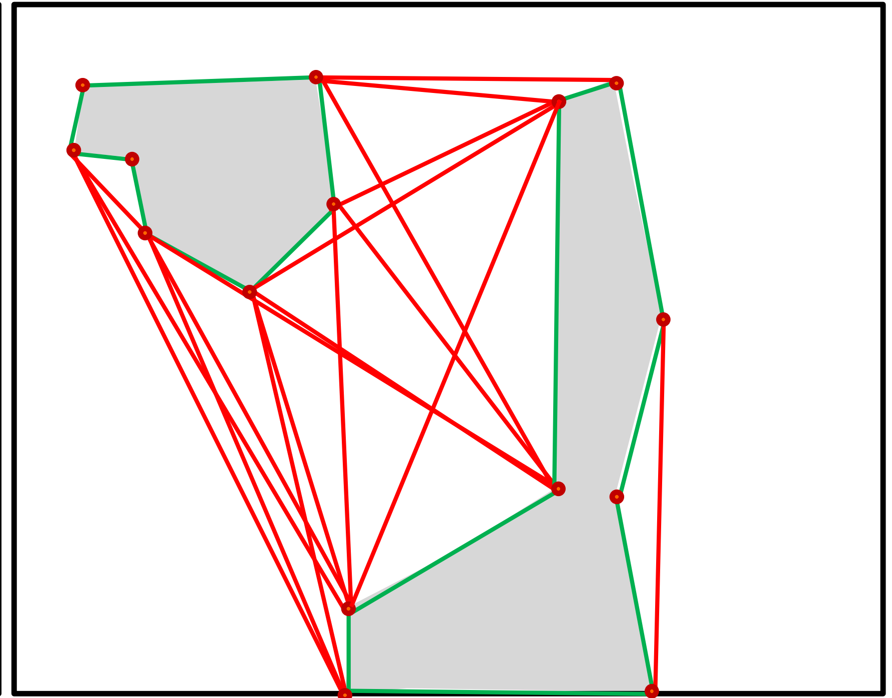
Remarks:

- definition applies only to convex obstacles:
- for concave polygons: compute the convex hull of each obstacle and add the additional edges
- The definition implies a naive $O(n^3)$ algorithm to construct a visibility graph. Computing a visibility graph can be optimized to $O(n^2)$ (O'Rourke 87*)*

Example: Visibility Graph



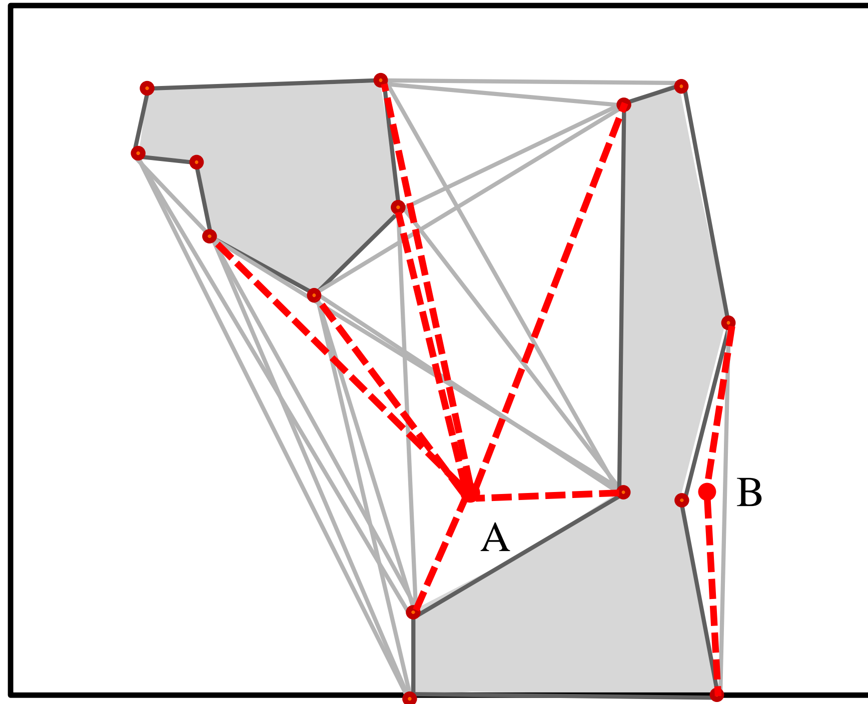
Edges for the node **A** being tested and discarded.



Visibility Graph: Red segments run between polygons. Green segments mark the polygons' borders.

Expansion with Start- and Goal-Nodes

- Visibility graph can be pre-calculated for static environments
- Mobile Objects must be integrated into the graph before calculation
- Inserting start S and goal Z as Point-Polygons
- Connecting the new nodes to with all edges unless an intersection with polygons occurs



Dijkstra's Algorithm

Used Data Structures:

- priorityqueue Q (stores found paths sorted by cost in descending order)
- nodetable T (contains cost for the currently best path for all visited nodes)

Pseudo-Code:

```
FUNCTION Path shortestPath(Node start, Node target)
    Q.insert(new Path(start,0))
    WHILE(Q.notIsEmpty())
        Path aktPath = Q.getFirst()
        IF aktPath.last() == target THEN                    //target found
            return aktPath
        ELSE
            FOR Node n in aktPath.last().successor() DO  //extend current path
                Path newPath = aktPath.extend(n)
                IF newPath.cost()<T.get(newPath.last()) THEN    //update optimal path
                    T.update(newPath.last,newPath.cost)
                    Q.insert(newPath,newPath.cost)
                ENDIF
            ENDDO
        ENDIF
    ENDWHILE
    RETURN NULL                                         //start and target not connected
ENDFUNCTION
```

A*-Search

- Dijkstra's algorithm uses no information about the direction to the target

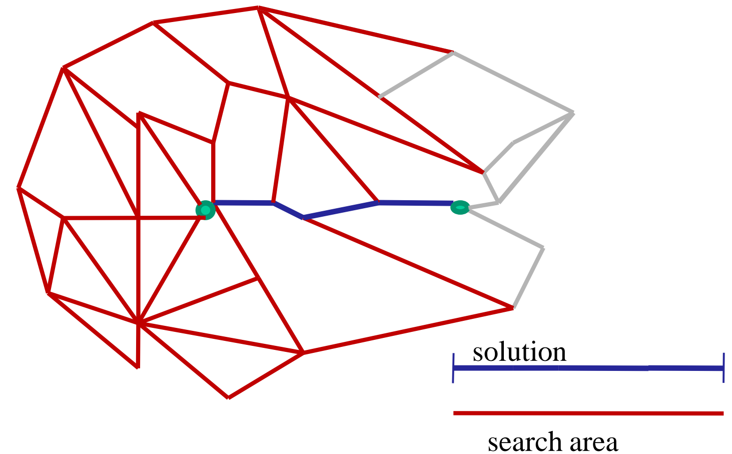
  => the search expands into all directions until the target is found
- A*-Search formalizes the direction into an optimistic forward approximation:

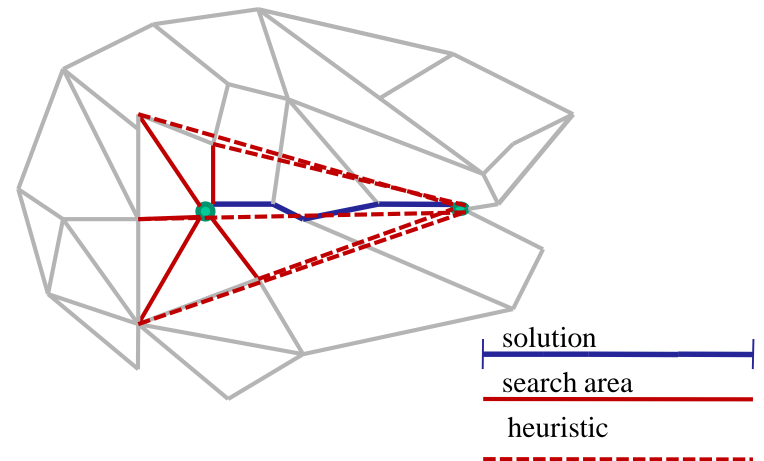
  h(n, target) for each node ***n***
- ***h(n,target)*** indicates a lower bound for the minimum cost to reach the target
- improve the search order by sorting by minimal total cost to reach the target
- allows to prune path ***P*** if:

  P.cost()+h(pfad1.last(),target) > bestPath.cost()

- basic heuristic for network distance:

  Euclidian distance between current position and target position.
  (a straight line is always the shortest connection)





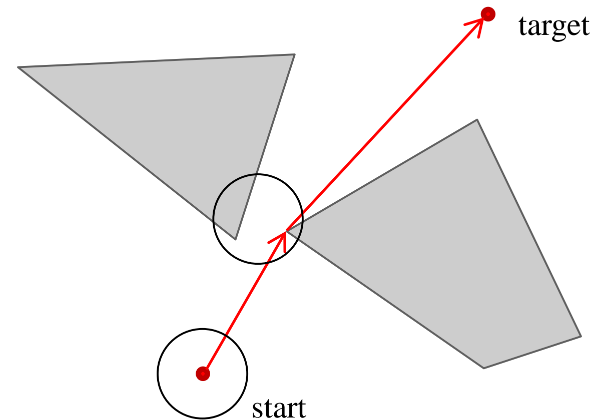
Pseudo-Code: A*-Search

Peseudo-Code: A*-Search

```
FUNCTION Path shortestPath(Node start, Node target)
 Q.insert(new Path(start),0)
  WHILE(Q.notIsEmpty())
      Path aktPath = Q.getFirst()
      IF aktPath.last()  == target THEN          //found result
         return aktPath
      ELSE
         FOR Node n  in aktPath.last().successor() DO    //expanding the current path
            Path newPath = aktPath.extend(n)
            IF newPath.cost()<T.get(newPath.last()) THEN //update if optimal so far
               T.update(newPath.last, newPath.cost())
               Q.insert(newPath, newPath.cost() +h(newPath.getLast(), target))
            ENDIF
         ENDDO
      ENDIF
 ENDWHILE
 RETURN NULL           //there is no path
ENDFUNCTION
```

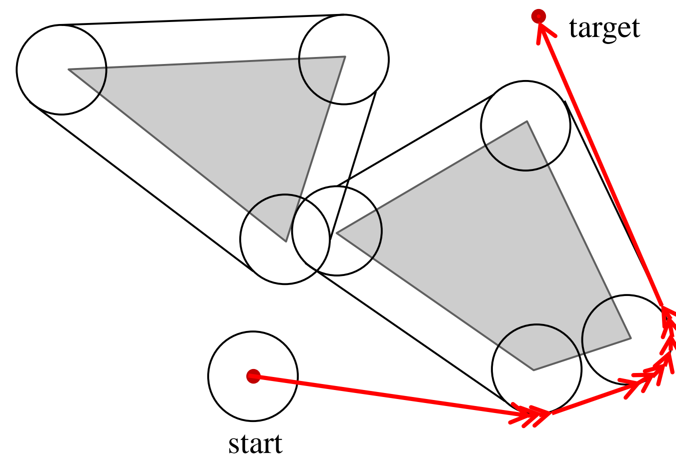
Visibility Graph for extended objects

- agents usually have a spatial expansion: Circle or Polygon
- visibility graph is only feasible for point objects
- adjust the visibility graph: expand obstacle polygons by the spatial expansion of the agent (Minkowski Sum)



problem with this solution:

- for circular expansion: circles have an infinite number of edges
  => visibility graph is not derivable
- For Polygon-Environment: object rotation should be considered
  => Every rotation requires a separate extension

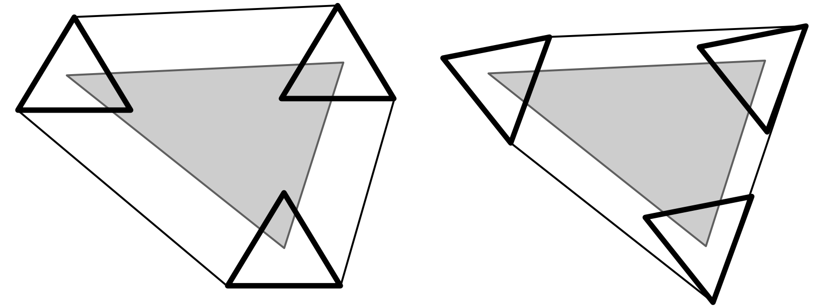


Visibility Graph for extended Objects

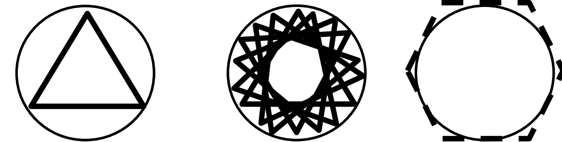
Solution Approach:

- polygons are approximated by the body of rotation => Circle
- circles are approximated by minimal surrounding polygons (MUP)

  => e.g. hexagon, octagon
- form Minkowski sum with the MUPs and derive visibility graph.



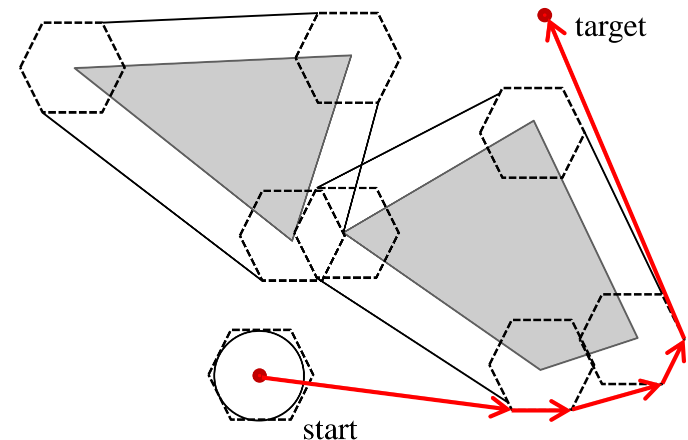
Minkowski sum of varying rotations of the same shape



double approximation by building the body of rotation and the minimal surrounding Hexagon

Remarks:

- paths are not optimal
- passages are considered conservative
- curves are taken angular
- MMO should only have a limited selection of agent extensions because each requires it's own graph



More Pathfinding Methods

Other Methods:

- approximate polygons with polygons having less corners
- hierarchic routing for longer routes
- precalculate and store shortest paths
- use a grid based graph, overlay the map with a grid and route over cell centers => decent approximation
- use sampled graphs in open environments (cmp. task-motion planning in robotics)

A
B

A
B

Markov Decision Process

Problem: We might not know the outcome of an executed action

- other players might act unpredictable
- game integrate random processes

$\Rightarrow$ **t(s,a) is stochastic**: $P(s'|s,a)$ for all $s,s' \in S$, $a \in A$

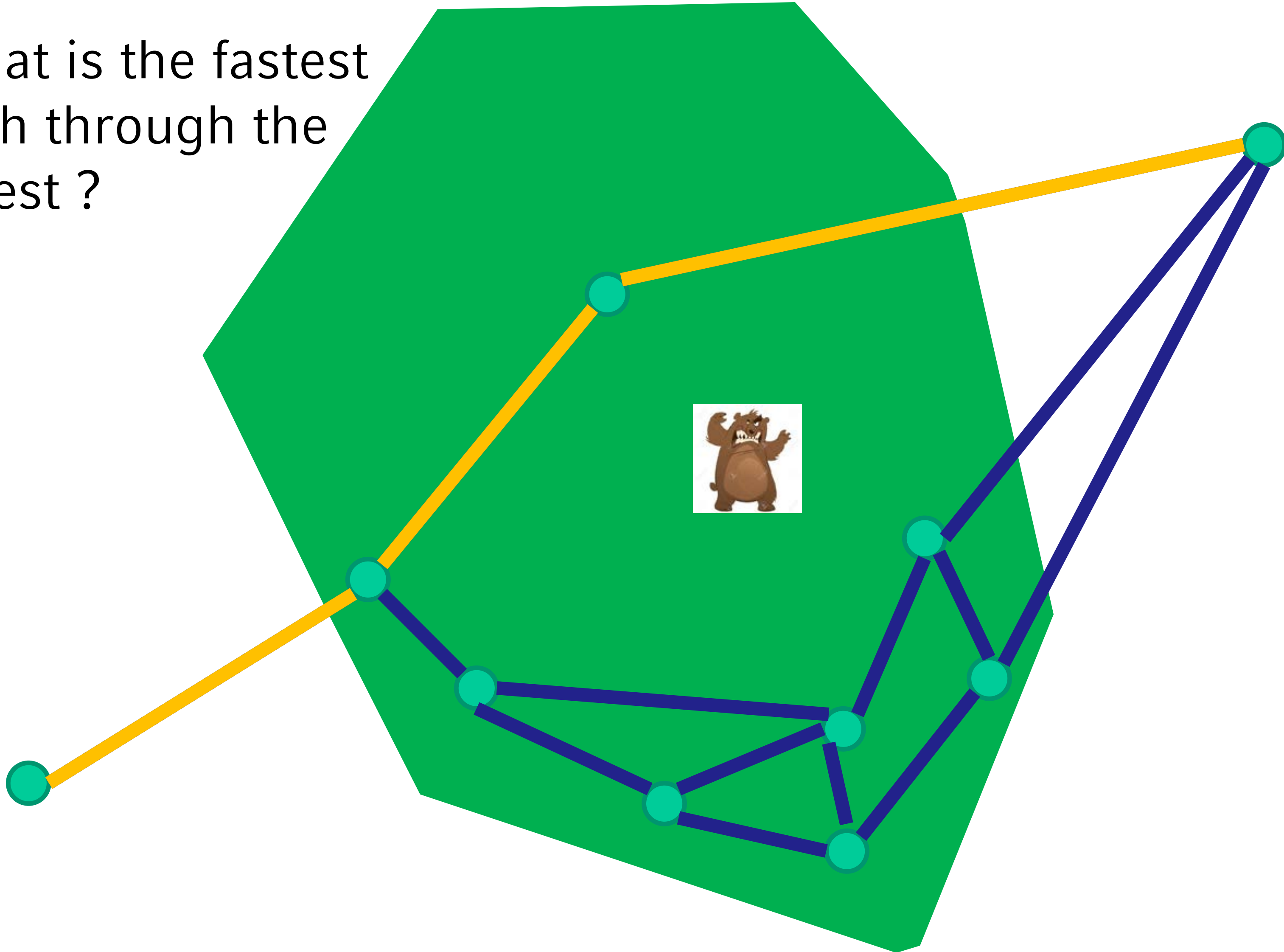
implications:

- we do not know what the future state after performing action a is.
- the reward R(s) gets stochastic as well
- when searching a terminal state, there is no secure path leading us to the target

=> We need an action for each situation we might encounter and not just the states on the path because we cannot control where we going to end up.

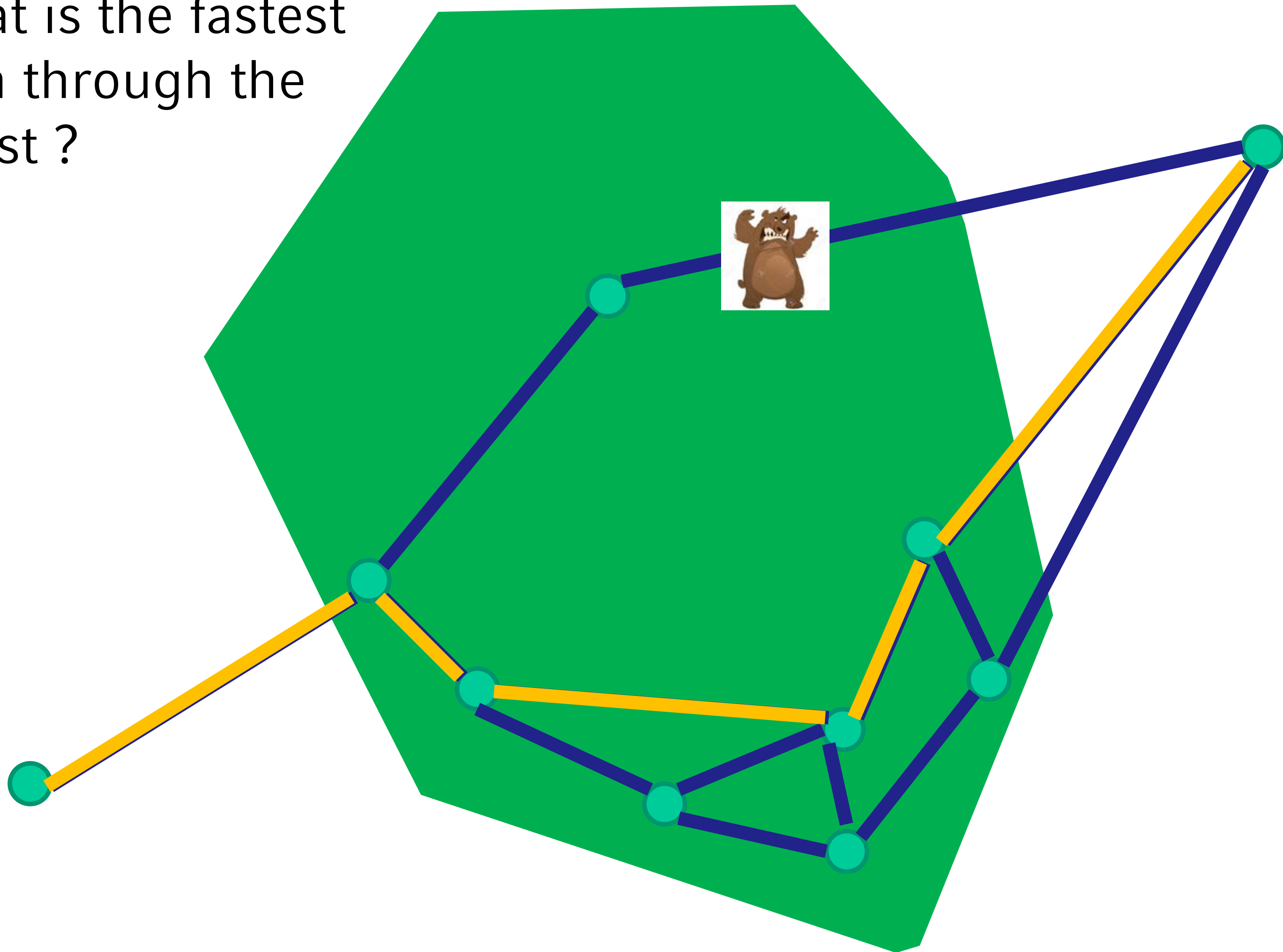
Motivation non-deterministic Routing

What is the fastest path through the forest ?



Motivation non-deterministic Routing

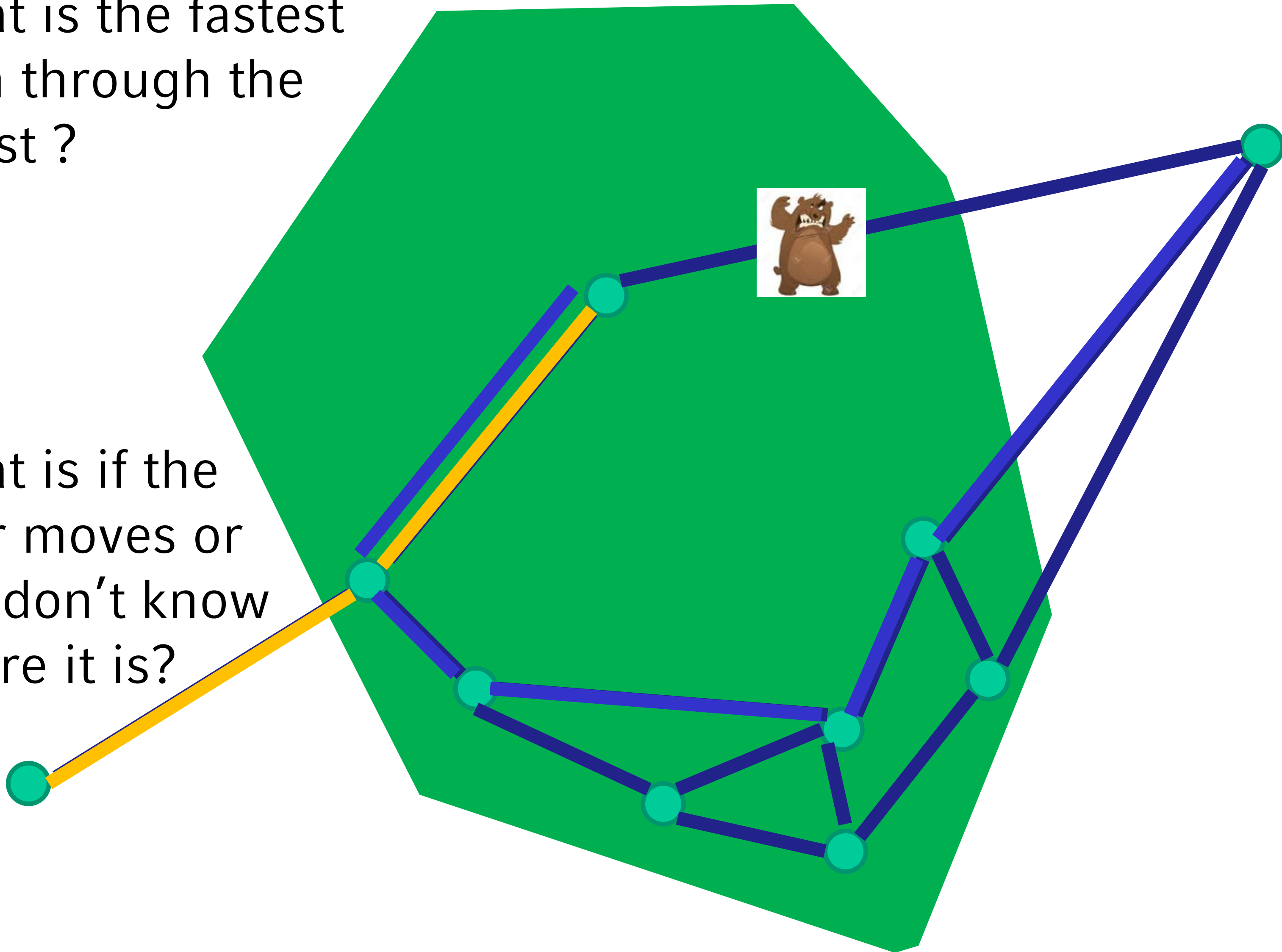
What is the fastest path through the forest ?



Motivation non-deterministic Routing

What is the fastest path through the forest ?

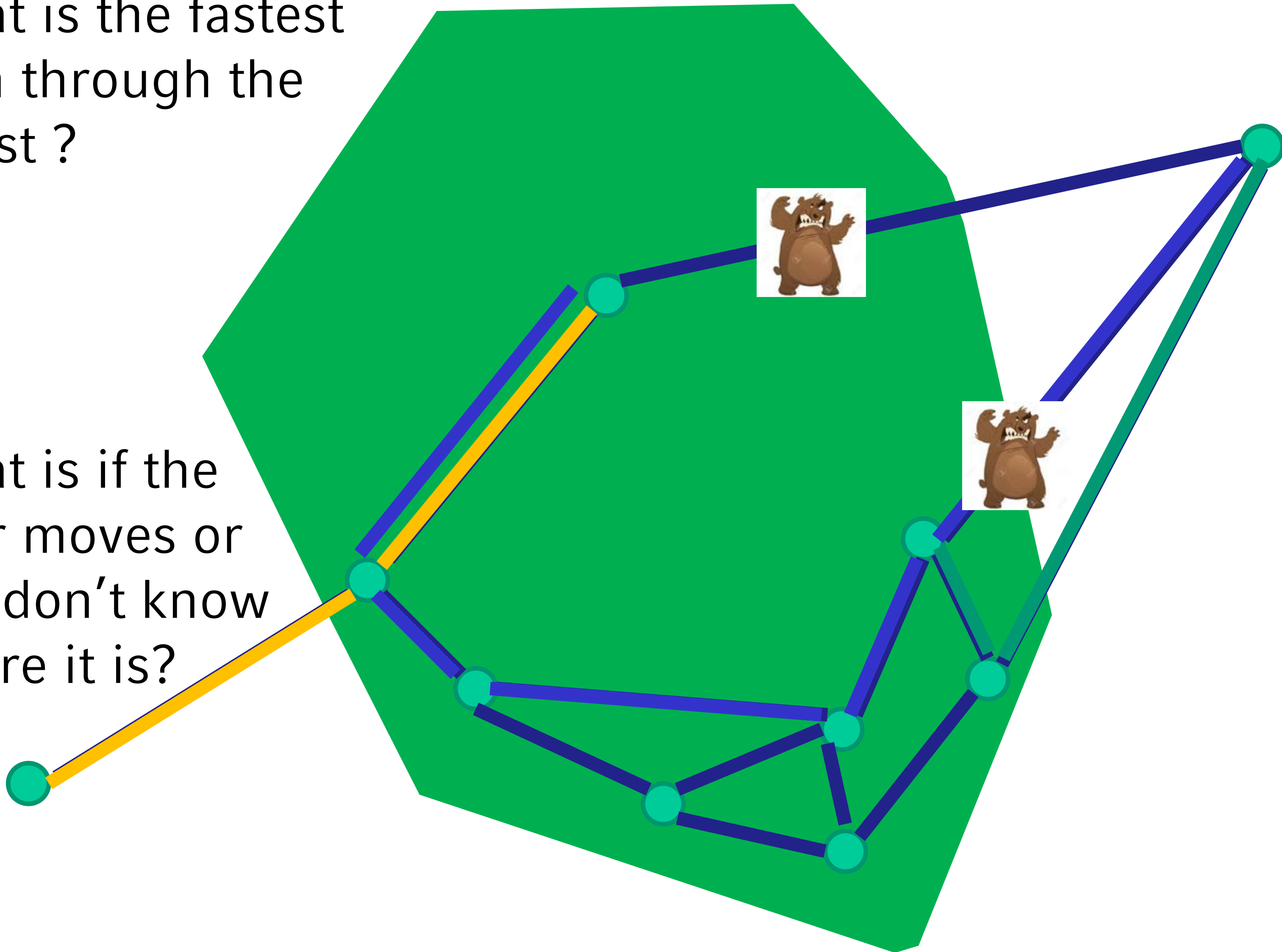
What is if the bear moves or you don't know where it is?



Motivation non-deterministic Routing

What is the fastest path through the forest ?

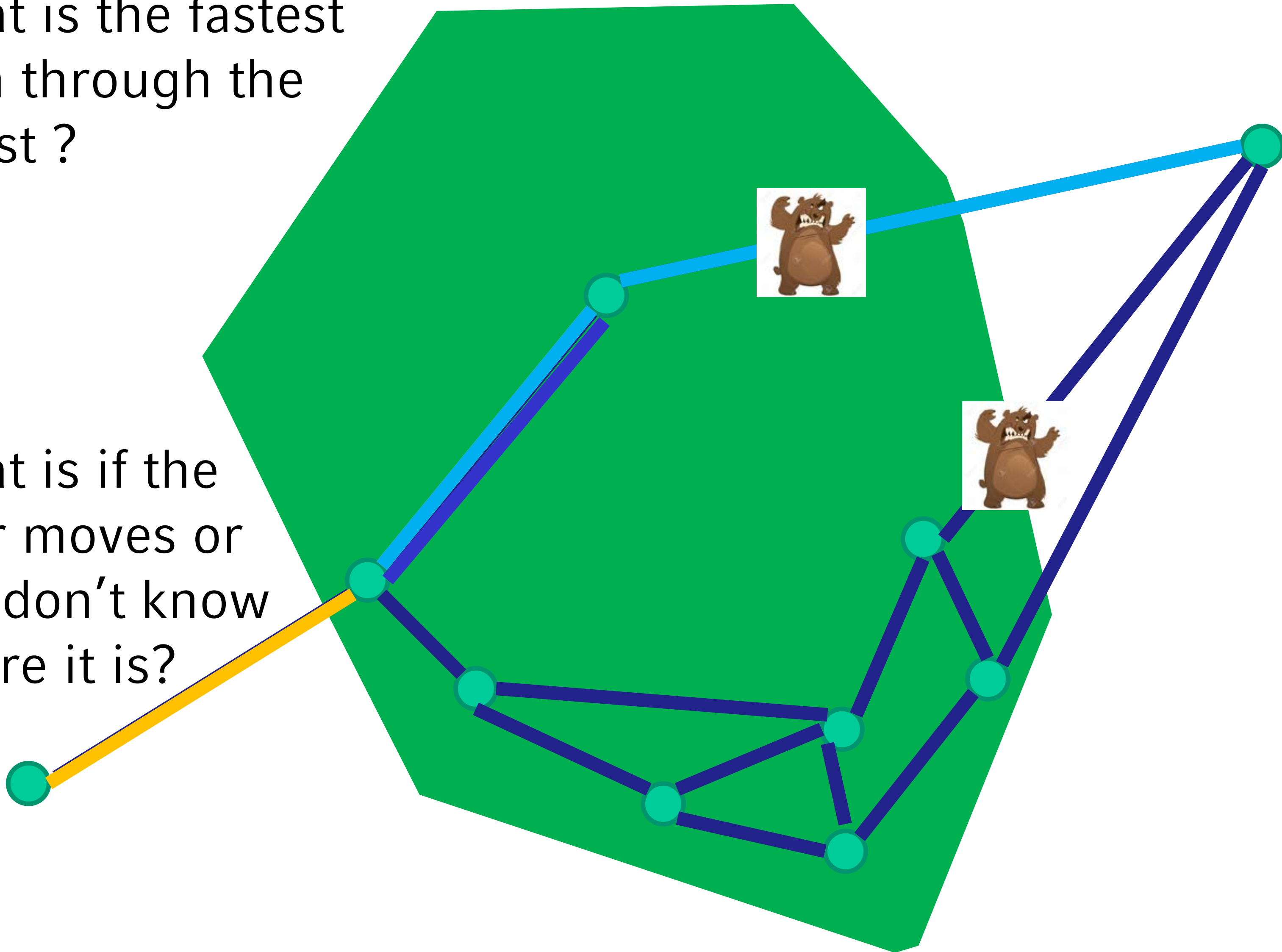
What is if the bear moves or you don’t know where it is?



Motivation non-deterministic Routing

What is the fastest path through the forest ?

What is if the bear moves or you don’t know where it is?



Policies and Utilities

Assume a **dynamic**, discrete, **non-deterministic**, known and fully observable environment.

- a policy π is a mapping defining for every state $s \in S$ an action $\pi(s) \in A(s)$ (agent knows what to do in any situation)
- Stochastic Policies: Sometimes it is beneficial to vary the action then $\pi(s)$ is a distribution function over A(s). (think of a game where strictly following a strategie makes you predictable)

Example:

Bellman's Equations

- What is the reward of following π?

  Utility $U^{\pi}(s)$: expected reward when following π in state s

  - $U^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t) \,|s_0 = s, \pi]$
  - $U^{\pi}(s) = \gamma^0 R(s) + E[\sum_{t=1}^{\infty} \gamma^t R(s_t) \,|s_0 = s, \pi]$
  - $U^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s, \pi) U^{\pi}(s')$

    (Bellmann Equation)

- What is the optimal policy π*?

  Bellman Optimality Equation:

$$U^{\pi*}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s, a)\, U^{\pi*}(s')$$

Finding optimal Policies: Policy Iteration

- We are looking for the optimal policy.
- The exact utility values are not relevant if one action is clearly the optimal.

- **Idea**: Alternate between:
  - **Policy evaluation**: given a policy, calculate the corresponding utility values
  - **Policy improvement**: Calculate the policy given the utility values $\pi(s) = \underset{a \in A}{argmax} \sum_{s' \in S} P(s'|s,a)\, U^*(s')$

Policy Evaluation

- Policy evaluation is much simpler than solving the Bellman equation

$$U^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s,\pi)\, U^{\pi}(s')$$

- Note that the non-linear function "max" is not present
- We can solve this by standard algorithms for linear equation systems.
- For large state spaces, solving systems of linear equations takes a long time ($O(n^3)$)
- In large state spaces a simplified Bellman update for k times can be more performant

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s,\pi_i)\, U_i(s')$$

Policy Iteration

repeat

$U \leftarrow \text{PolicyEvaluation}(\pi, U, MDP)$

$unchanged? \leftarrow true$

for each state s in S **do**

if $\max_{\boldsymbol{a} \in \boldsymbol{A}} \sum_{\boldsymbol{s}'} \boldsymbol{P}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})\, \boldsymbol{U}[\boldsymbol{s}'] > \sum_{\boldsymbol{s}'} \boldsymbol{P}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{\pi})\, \boldsymbol{U}[\boldsymbol{s}']$ **then**

$\boldsymbol{\pi}[\boldsymbol{s}] \leftarrow \operatorname{argmax}_{a \in A} \sum_{\boldsymbol{s}'} \boldsymbol{P}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})\, \boldsymbol{U}[\boldsymbol{s}']$

$\boldsymbol{unchanged}? \leftarrow \boldsymbol{false}$

until $unchanged?$

return π

Value Iteration

- if we use Bellman updates anyway, we can join both steps
- update Bellman optimality equation directly:

$$U^*(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a)\, U^*(s')$$

- Non-linear system of equations
- Use Dynamic Programming
  - Compute utility values for each state by using the current utility estimate
  - Repeat until it converges to U^*
  - convergence can be shown by contraction

Value Iteration

repeat

$U \leftarrow U'$

$\delta \leftarrow 0$

for each state s in S **do**

$U'[s] \leftarrow R(s) + \gamma \max_{a \in A} \sum_{s'} P(s'|s,a)\, U[s']$

if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$

until $\delta < \frac{\epsilon(1-\gamma)}{\gamma}$

return U

MDP Synopsis

- MDPs rely on a Markov model with assumptions about: states, actions, rewards, transition probabilities, etc.
- if all the information is available, computing the optimal policy does not require any learning samples
- Transitions probabilities are usually not defined but have to be estimated based on observations
- usually observations $\neq$ states
  - $\Rightarrow$ partially observable MDP, estimate belief states (compare HMM)
  - $\Rightarrow$ set of possible states and transitions is unknown

Can we learn on observations only without making an model assumptions ?

Model-Free Reinforcement Learning

If we don't have a model, what do we have:

NOTE: s is a state description, but S does not need to be known.

1. Sample Episodes:
   - episode = $s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, s_3 \ldots, s_l, a_l, r_l, s_{l+1}$
   - *reward of the episode:* $\sum_{i=1}^{l} \gamma^i r_i$ *with* $0 < \gamma \leq 1$
   - episode might end with terminal state
2. Queryable Environments:
   - Agent selects an action $a \in A(s)$ and receives on new state s', R(s'), A(s') from the Environment.
   - Allows to generate episodes

Monte-Carlo Policy Evaluation

- for a known policy π and a set of sample episodes X following π
- let X(s) be the set of (sub-)episodes starting with s
- to estimate utility $U^{\pi}(s)$ average over the expected reward:

$$\mathrm{U}(s) = \sum_{x \in X(s)} \frac{\sum_{i=1}^{l} \gamma^i r_i}{\lfloor X(s) \rfloor}$$

- if X(s) gets sufficiently large for all $s \in S$: $U(s) \rightarrow U^{\pi}(s)$
- if new episodes arrive, compute incremental mean:

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j = \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) = \frac{1}{k}(x_k + (k-1)\mu_{k-1})$$

$$= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

- if the environment is non-stationary, limit the weight of old episodes:

$$\mathrm{U}(s_t) \leftarrow \mathrm{U}(s_t) + \alpha\big(R(x) - \mathrm{U}(s_t)\big)$$

Temporal Difference Learning

problem: Can we still learn if episodes are incomplete?

- the later part of $\sum_{i=1}^{l} \gamma^i r_i$ is missing
- in the extreme case we just have 1 Step: s_t, a, r, s_{t+1}

=> Temporal Difference Learning

- idea similar to incremental Monte-Carlo learning:

$$\mathrm{U}(s_t) \leftarrow \mathrm{U}(s_t) + \alpha\big(R(x) - \mathrm{U}(s_t)\big)$$

- Policy Evaluation with Temporal Difference (TD) Learning:

$$\mathrm{U}(s_t) \leftarrow \mathrm{U}(s_t) + \alpha\big(R(s_{t+1}) + \gamma\mathrm{U}(s_{t+1}) - \mathrm{U}(s_t)\big)$$

- TD target: $R(s_{t+1}) + \gamma\mathrm{U}(s_{t+1})$
- TD error: $R(s_{t+1}) + \gamma\mathrm{U}(s_{t+1}) - \mathrm{U}(s_t)$
- each step estimates the mean utility incrementally

Policy Optimization

Idea: adapt Policy Iteration
(evaluate policy and update greedily)

- greedy policy update of U(s) requires a MDP:

$$\pi'(s) = argmax_{a \in A(s)} P(s'|s,a) U(s')$$

- Q-Value Q(s,a): If we choose action a in state s
  what is the expected reward?
  => We do not need to know where the action will take us!
- Improving Q(s,a) is model free:

$$\pi'(s) = argmax_{a \in A(s)} Q(s,a)$$

- Adapt the idea of Policy Iteration:
  - Start with a default policy
  - evaluate policy (previous slide)
  - update policy: e.g. with greedy strategy

Samples and Policy Updates

Problem: After updating a policy, we need enough samples following the policy.

- real observed episodes usually do not cover enough policies (episodic samples are policy dependent)

$$s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, s_3 \ldots, s_l, a_l, r_l, s_{l+1}$$

- we need to dynamically sample from an environment:
  - measure reaction of physical world (e.g. robotics..)
  - build simulations which mimics the physical world
  - in Games: let the agent play and learn !!!
- We need a strategy for sampling these s,a pairs.
- s is often determined by the environment as result of the last action. (The game is in state s after the last move.)

Learning on a Queryable Environment

- we can generate as much samples as possible
- environment might be non-deterministic:
  - same state s and action a => different outcomes s' and R(s')
  - multiple samples for the same (s,a) might be necessary

- How to sample over the state-action space?
  - **exploit**: If we find a good action keep it and improve the estimate of Q(s,a). Usually, it's a waste of time to optimize Q(s,a) for bad actions.
  - **explore**: Select unknown or undersampled actions
    - a low Q(s,a) need not mean that the option is bad, maybe it is just underexplored.
    - try out new things might lead to a even better solution

ε-Greedy Exploration

- makes sure that sampling considers new actions
- when sampling:
  - With probability 1-ε choose greedy action
  - with probability ε chose random action
- Sampling policy:

$$\pi(a|s) = \begin{cases} \frac{\varepsilon}{m} + (1 - \varepsilon) & if\ a = argmax_{a \in A(s)} Q(s, a) \\ \frac{\varepsilon}{m} & otherwise \end{cases}$$

- achieves that Q-values improve and guarantees that all actions are explored if optimized long enough

On-Policy and Off-Policy Learning

Which Q(s,a) is used for sampling an action?

on-policy learning: Sample with respect to the currently learned policy. ***example***: SARSA

off-policy learning: Sampling is done based on a behavioral policy which is different from the learned policy.

example: Q-Learning

Implication:

- For off-policy learning, exploration is usually just done for the behavioral policy.
- For on-policy methods, exploration must be part of the learned policy.

Q-Learning

- standard off-policy learning method
- Given a behavioral policy π_b, learn the policy π_l by the following learning update:

$$Q(s, \pi_b(s)) \leftarrow Q(s, \pi_b(s)) + \alpha\big(R(s) + \gamma\, Q(s', \pi_l(s')) - Q(s, \pi_b(s))\big)$$

- Usually we want to learn the optimal policy, thus:

$$\pi_l(s) = \operatorname*{argmax}_{a \in A} Q(s, a)$$

- For behavioral policy, choose ε-Greedy

Q-Learning Algorithm

```
init Q(s,a)∀s ∈ S,a ∈ A
for n episodes:
  init s
  repeat until episode is finished:
    choose a from A(s) with π_b
    s',r = query_Env(s,a)
    Q(s,a)← Q(s,a)+α(r+γmax_a Q(S',a)-Q(s,a))
    s ← s'
  until s is terminated
//terminal state or finite horizon is reached
```

Function Approximation of State Spaces

- Q-Learning collects Q-Values for all explored state-action pairs (s,a) => Q-Learning maintains a Q-table
- Is the state of observation the state space for making decision?
  - state spaces are often exponential in the number of variables
  - similar states usually require similar actions
- basic Q-Learning does not generalize from observations to states

Idea: Function Approximation

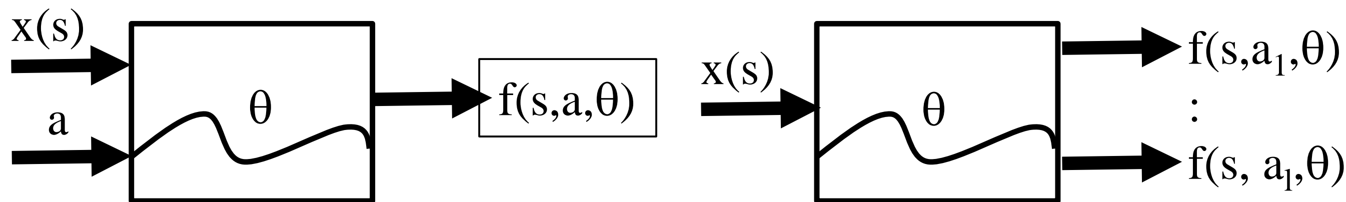
Treat the set of states as a (continuous) vector of factors and learn a regression function $f(s,a,\theta)$ predicting $Q^*(s,a)$.

Q-value function approximation

Given: A mapping *x(s)* describing *s* in IR^d.

Goal: Learn a function $f(x(s),a,\theta)$ predicting the true Q-value Q*(s,a) for any value of x(s).

- similar to supervised learning, but not exactly:
  - Where to put the action *a* in our prediction function?



  - Samples from the same trajectory are not independent and identical distributed (IID)
  - true Q*(s,a) is not known for training

    => targets are constantly changing

Learning using Function Approximation

- we want to learn a function f(x(s),a,θ) over the state-action space by optimizing the function parameters θ.

$$f(x(s),a,\theta) \approx Q^*(s,a)$$

- to learn f we need a loss function, e.g. MSE between $f(s,a,\theta)$ and observed values Q*(s,a).

$$L(\theta) = E\left[\left(Q^*(s,a) - f(x(s),a,\theta)\right)^2\right]$$

- optimization using stochastic gradient descent

$$-\frac{1}{2}\nabla L(\theta) = \left(Q^*(s,a) - f(x(s),a,\theta)\right)\nabla_\theta f(x(s),a,\theta)$$

$$\Delta\theta = \alpha\left(Q^*(s,a) - f(x(s),a,\theta)\right)\nabla_\theta f(x(s),a,\theta)$$

- update: $\theta \leftarrow \theta + \Delta\theta$

Linear Prediction Functions

A simple function approximation might be linear

- Linear Functions over $s \in IR^d$:

$$f(x(s), a, \mathrm{W}) = \mathrm{x(s)^T W} = \sum_{j=1}^{n} x(s)_j \; w_j$$

- Loss function:

$$L(W) = E[(Q^*(s,a) - \mathrm{x(s)^T W})^2]$$

- Stochastic Gradient Descent on L(w):

$$\nabla W \; f(x(s), a, \mathrm{W}) = \mathrm{x(s)}$$

$$-\frac{1}{2} \nabla L(\theta) = \left(Q^*(s,a) - f(x(s), a, \theta)\right) x(\mathrm{s})$$

$$\Delta\theta = \alpha\left(Q^*(s,a) - f(x(s), a, \theta)\right) x(\mathrm{s})$$

Further Directions

- other prediction functions:
  - (deep) neural networks
  - decision trees
  - nearest neighbor
  - ...
- DQN: uses a deep neural network and works with an experience buffer to make the learning target more stable
- Policy Gradients: Uses function approximation for selecting the best action (not the Q-values)
- Actor-Critic methods: Combine value function approximation and policy gradient.

Why is AI important for Games?

Computer games are an optimal sand-box for developing AI techniques:

- games are queryable environments
- rewards and actions are known
- states are parts or views on the game state

But, why is reinforcement learning interesting for managing and mining Computer Games ?

- develop intelligent AI opponents/collaborators
- micro-management for small granularity games
- learn optimal strategies for teaching players or balancing
- mimic real behavior within a game

Imitation Learning

- use reinforcement learning to make an agent behave like a teacher (e.g. a pro gamer)
- Learning from experience: teacher provides (s,a,r,s') samples of good behavior (reward is known)
- Learning from demonstration: teacher provides (s,a,s') samples.
  - reward is not explicitly known
  - success is expected based on the reputation of the player

Challenge:

- predicting the action for states with sufficient samples is easy (policy follows the distribution of observed actions)
- predicting proper actions for undersampled states is hard.

=> approximation function must generalized from observed states to unobserved ones.

Imitation learning in Games

possible applications:

- make a player behave like a real one
  (e.g. adapt player styles for football games)
- learn policies for hard opponents to analyze their weaknesses
- when training an agent learn from human experts
  (first Alpha Go version)
- learn policies for your own behavior and find out where it deviates from the optimal policy

Note, this is an active field of research with many unsolved problems:

- policies depend on the agents/players capabilities
- capability of the imitating agent in unknown states is hard to evaluate
- reward functions might not be the same for teacher and imitating agent

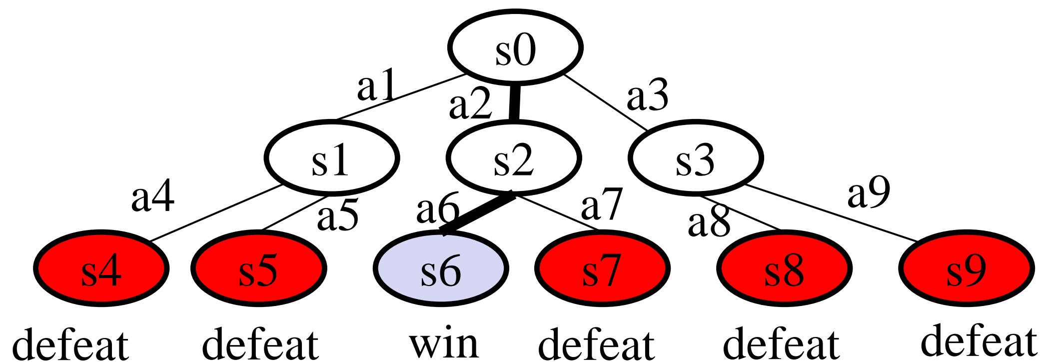
Techniques for Multiple Agents

Consider an MDP (S,A,T,R):

- often the uncertainty of state transitions T is completely caused by the actions of other independent agents (opponent or team members)

  examples: chess, GO, etc.

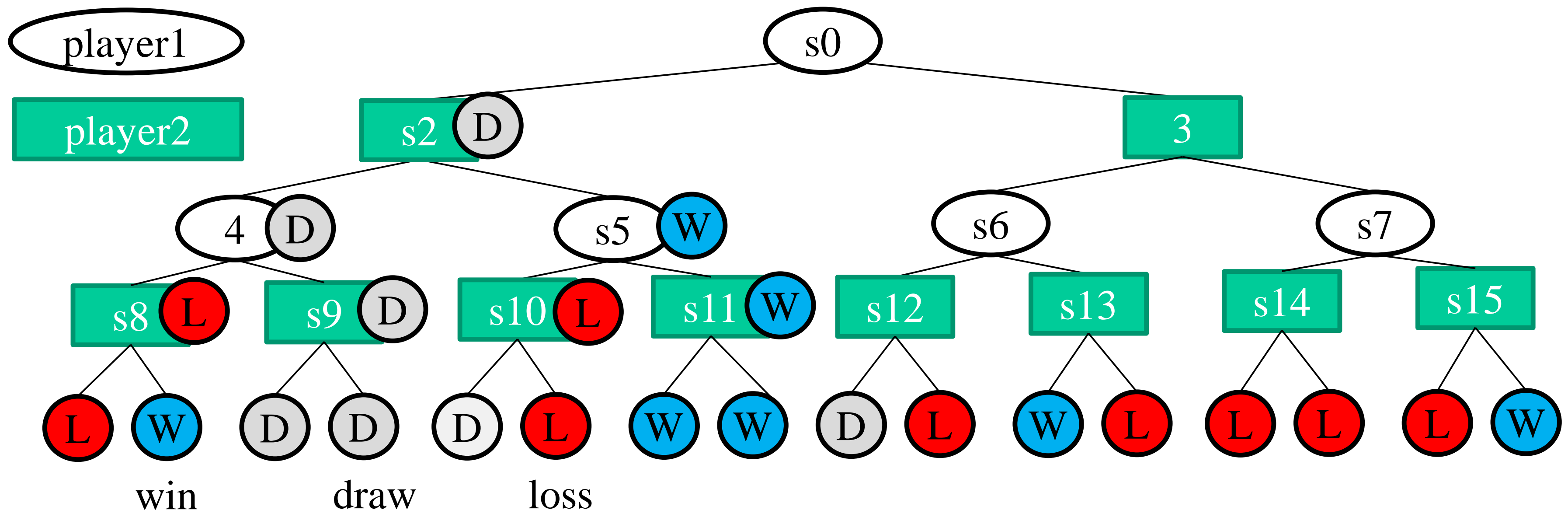
- if you would know the policy of the other agents, optimal game play could be achieved with deterministic search.



Antagonistic Search

- assume that there is a policy π^* which both player follows
- in antagonistic games, the reward of player p1 is the negative reward of player p2. (zero-sum game)

=> player1 maximizes rewards
player2 minimizes the rewards



Antagonistic Search

- generally it is not possible to search until the game ends (search grows exponential with available actions)

$\Rightarrow$ stop searching at a certain level and user another reward corresponding to the chance of success

Types of rewards:

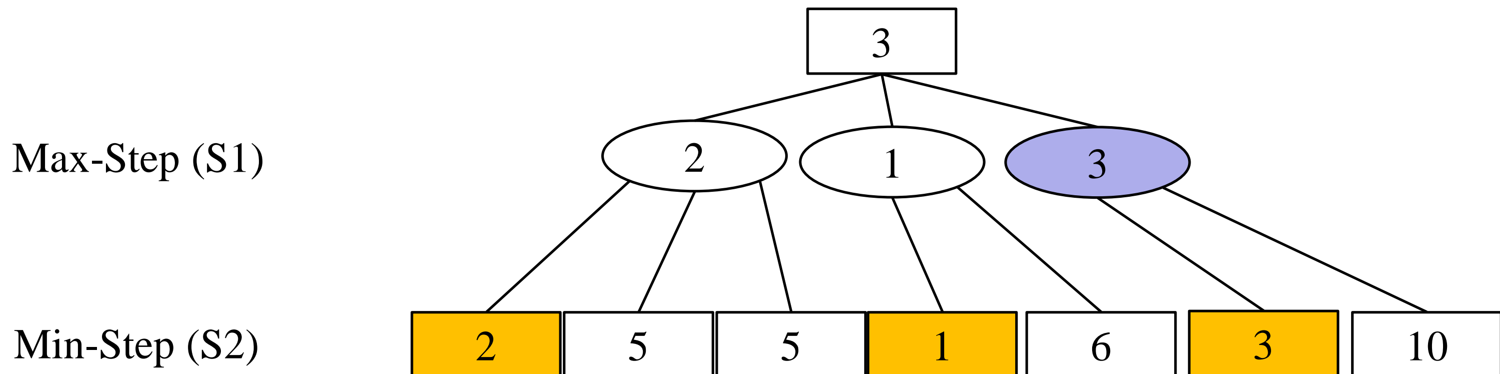
- heuristics (figures, flexibility, strategic positions etc.)
- prediction functions (input game state ->win probability)
- databases (opening or end game libraries)

Min-Max Search in antagonistic Search Trees

- select action a that maximizes R(s) for S1 after S2's reaction
- Search depth:
  - Given Number of Turns

  ⇒ Time may vary and is hard to estimate

  ⇒ Turbulent positions make cutting of some branches unfavorable
  - Iterative Deepening:
    - Multiple calculations with increasing search depth
    - On Time-Out: Abort and use of last complete calculation
      (since expense doubles on average, double the expense can be estimated)
  - turbulent positions: single branches are being expanded if leaves are turbulent.



Alpha-Beta Pruning

Idea: If a move already exists, that can be valuated with even after a counter reaction, all branches creating a value less than can be cut.

- : S1 reaches at least α on this sub-tree ($R(s) > \alpha$)
- : S2 reaches at most β on this sub-tree ($R(s) < \beta$)

Algorithm:

- Traverse Search-Tree with deep search and fill inner nodes on the way back to the last branching
- For calculating inner nodes:

  If $\beta < \alpha$ then

  - Cut off remaining sub-tree
  - set β-value for the sub-tree if it's root is a min-node
  - set α -value for the sub-tree if it's root is a max-node

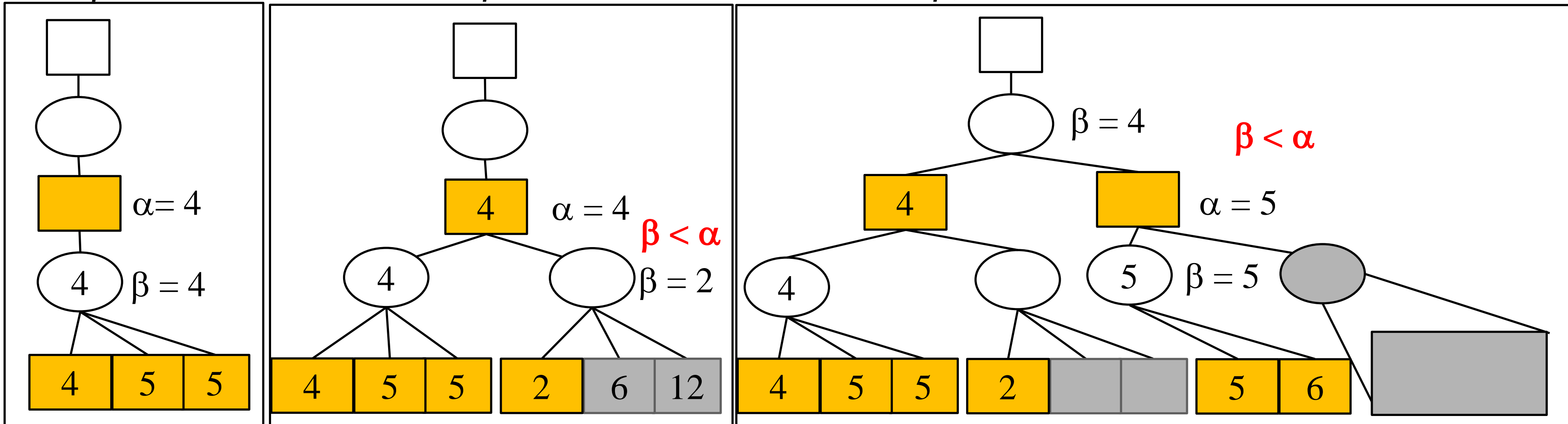
  Else set β-value to the minimum of min-nodes

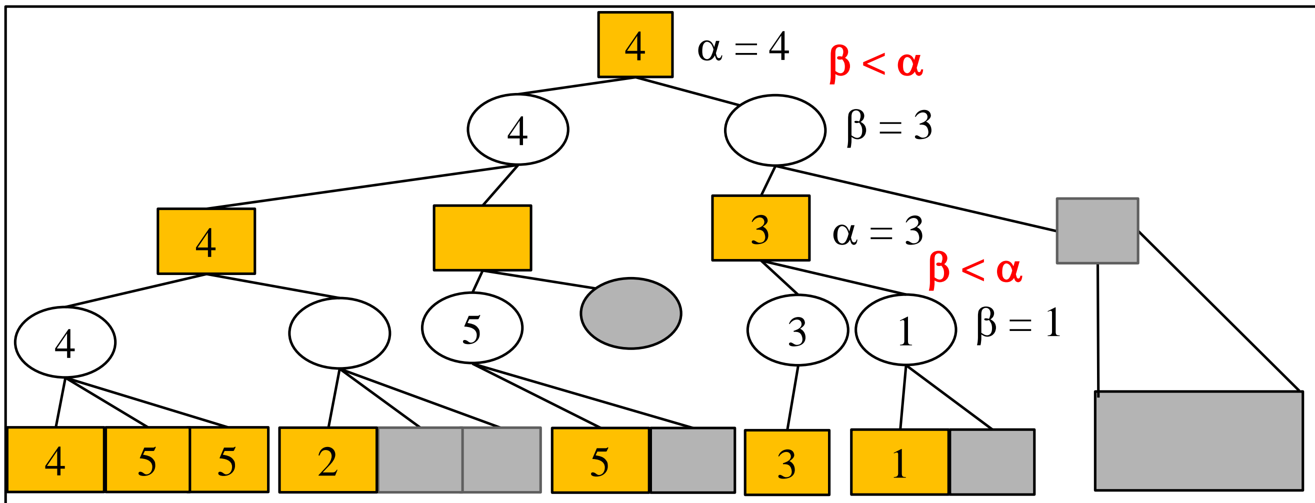
  set α-value to the maximum of max-nodes

Alpha-Beta Pruning

Idea: If a move already exists, that can be valuated with α even after a counter reaction, all branches creating a value less than α can be cut.

- α: S1 reaches at least α on this sub-tree ($R(s) > \alpha$)
- β: S2 reaches at most β on this sub-tree ($R(s) < \beta$)





Monte Carlo Tree Search

- for games with high branching factors MinMax does not scale
- heuristics are often hard determine and require expert knowledge
- machine learning depends on the available data sets (biased to human play style)

Monte Carlo Tree Search:

- samples tree based on Monte Carlo Learning of simulated play outs
- uses an exploration/exploitation scheme to systematically search the first k-layers of the search tree.
- simulation can be based on different opponent agents strategies

UBC1

- selects actions w.r.t. reasonable exploration and exploitation trade-offs
- consider a situation where you had N tries and l actions
- for each action a_i you know the number of wins and number of samples (allows to calculate mean win rate)
- based on Hoeffding's inequality, it can be shown that the following bound for mean win rate holds: $c_{n,n_i} = \sqrt{\frac{2 \ln n}{n_i}}$
- the bounds gets narrower the more samples for a_i become available, but the bounds for all actions aj (i≠j) become wider
- now always select action $a = argmax_i(\mu_i + c_{n,n_i})$

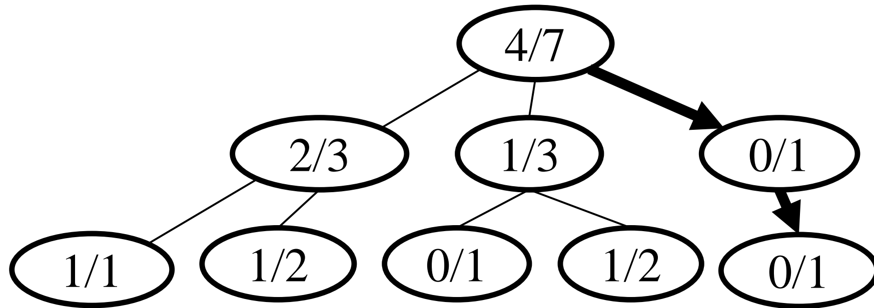
Monte Carlo Tree Search with UCT

- use UBC1 for sampling the first k levels of the search tree
- if no samples are available apply a random search or some light-weight policy.
- to evaluate leafs at the leaf level, simulate game until terminal state is reached

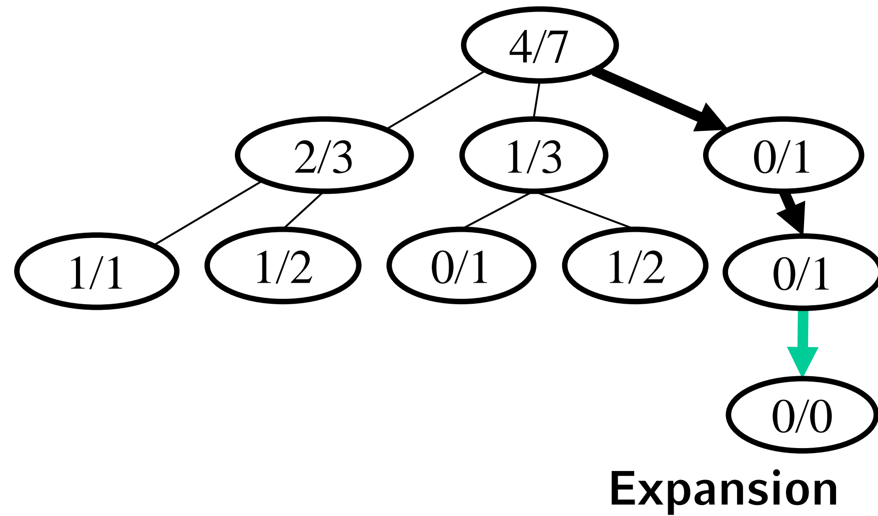
The algorithm runs in 4 phases:

- selection: search tree based on UBC1
- expansion: randomly select an action when UBC1 does not work
- simulation: simulate a further game trajectory
- backpropagation: backup the value along the path to the root

Example

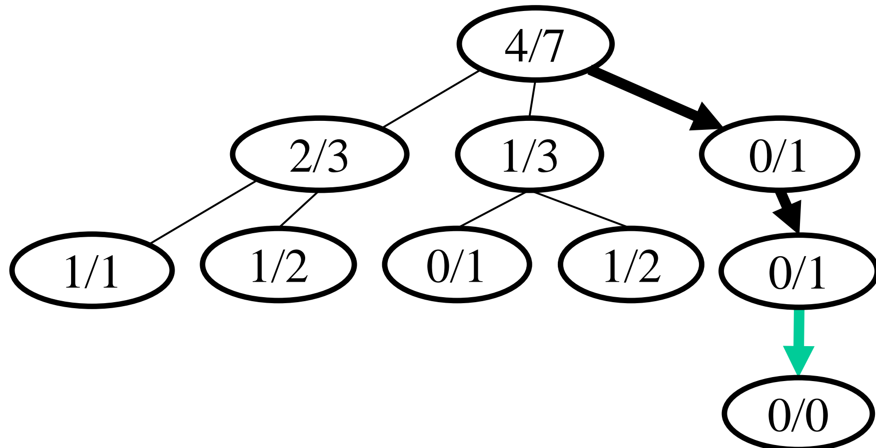


Selection

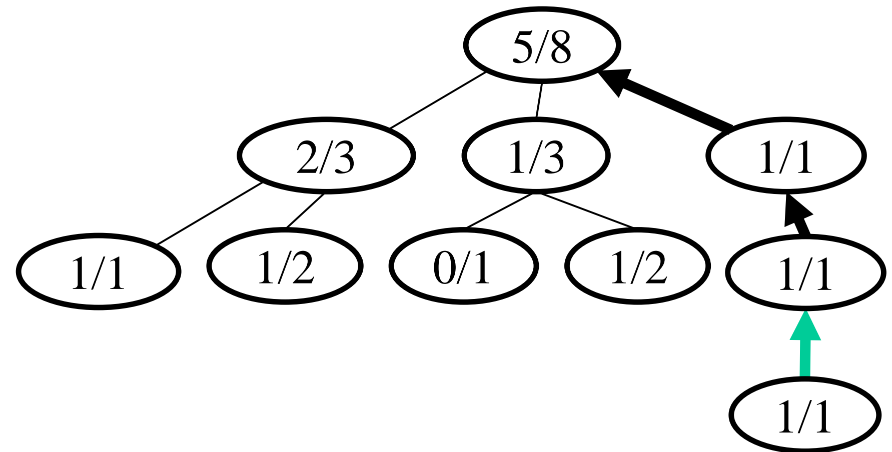


Expansion

Example



Simulation



Backpropagation

Monte Carlo Tree Search

- applicable to antagonistic search but not restricted to it
- can handle stochastic games and games partially observable game states
- the 4 steps can be iterated until a given time budget is spend: the longer the search is done the better is the result.
- a general question is to perform simulation to determine the possible outcomes
- Monte Carlo Tress Search is used in Alpha Go to allow lookahead together with convoluational neural networks and deep reinforcement learning

Modern Superhuman Game Agents

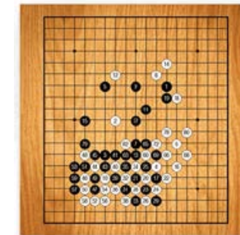
- superhuman = better than the best human players
- mostly showroom projects for pushing the limits of AI
- making AI to behave coordinated, goal oriented and general

Examples:

- Atari Games
  (various Atari 2600 games learned by an identical network by DeepMind)
- AlphaGO and AlphaGO Zero
  (first GO-AI beating a grand Master 9th Dan & successor for chess, Shogi and Go by DeepMind)
- OpenAI 5
  (Dota2 team agent competing against E-Sports teams by OpenAI)
- AlphaStar (Starcraft2 AI for Protoss vs. Protoss by DeepMind)
- Capture the Flag: Quake III Arena by DeepMind
- …

Reinforcement Leaning for board games

- games are classical domain of AI because opponent behaves not just nondeterministic but antagonistic
- progress on games like chess, backgammon, Poker.
- for a long time Go was considered the biggest challenge due to its enormous branching factor
- In Jan 2016: Deepmind challenged Grand Maste Lee Sedol with a program called Alpha Go and won.
- Alpha Go is a combination of reinforcement learning and Deep learning techniques which was trained on a huge database of Go matches.
- In 2017 the same team proposed Alpha Go Zero: Learns any board Game based on the rules only by playing against version of itself (evaluated on Chess, Shogi and Go). Alpha Go Zero outperformed all specialized top AIs without imitating human game play.



Captured Stones

3 hours

AlphaGo Zero plays like a human beginner, forgoing long term strategy to focus on greedily capturing as many stones as possible.

19 hours

AlphaGo Zero has learnt the fundamentals of more advanced Go strategies such as life-and-death, influence and territory.

Captured Stones

70 hours

AlphaGo Zero plays at super-human level. The game is disciplined and involves multiple challenges across the board.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. & Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, 550, 354--.

Games as Testbed for AGIs

AI is not build single purpose but receives images and general controls such as a human.

- DQN on Atari Games

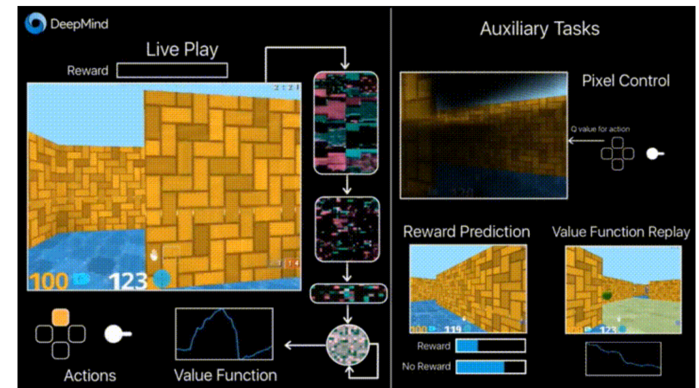
https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/

- OpenAI Five (Dota2)

https://blog.openai.com/openai-five/

- RL for Starcraft II:

https://deepmind.com/blog/deepmind-and-blizzard-open-starcraft-ii-ai-research-environment/



https://techxplore.com/news/2016-11-deepmind-boost-ai-unreal-agent.html

Architectures

- Complex systems being developed by expert teams
- Subtasks:
  - Learning a state representation approximating a Markov state from a sequence of agent observations
  - Value function approximation to asses the win probability for a state
  - Action generation to select a suitable action or multiple actions
  - Search for the next best move to include deterministic parts
  - …

Many of these tasks are currently solved with Deep Neural Networks for sequence modelling and Reinforcement learning

Experience Sampling

- Imitation Learning:
  - use experience from human players
  - reward the agent for making the same decision as humans (play like Ronaldo and not shoot more goals !!!)
  - important to learn meaningful actions
  - limited to the availability of replays
- Selfplay
  - Agents gathers experience by playing against itself
  - Rules are sufficient
  - Allows to generate new experience for new policies in short time
  - Allows to adjust opponent skill
    - challenging: winning requires non-trivial strategy
    - not frustrating: opponent let the agent gather experience to evolve, allows positive feedback

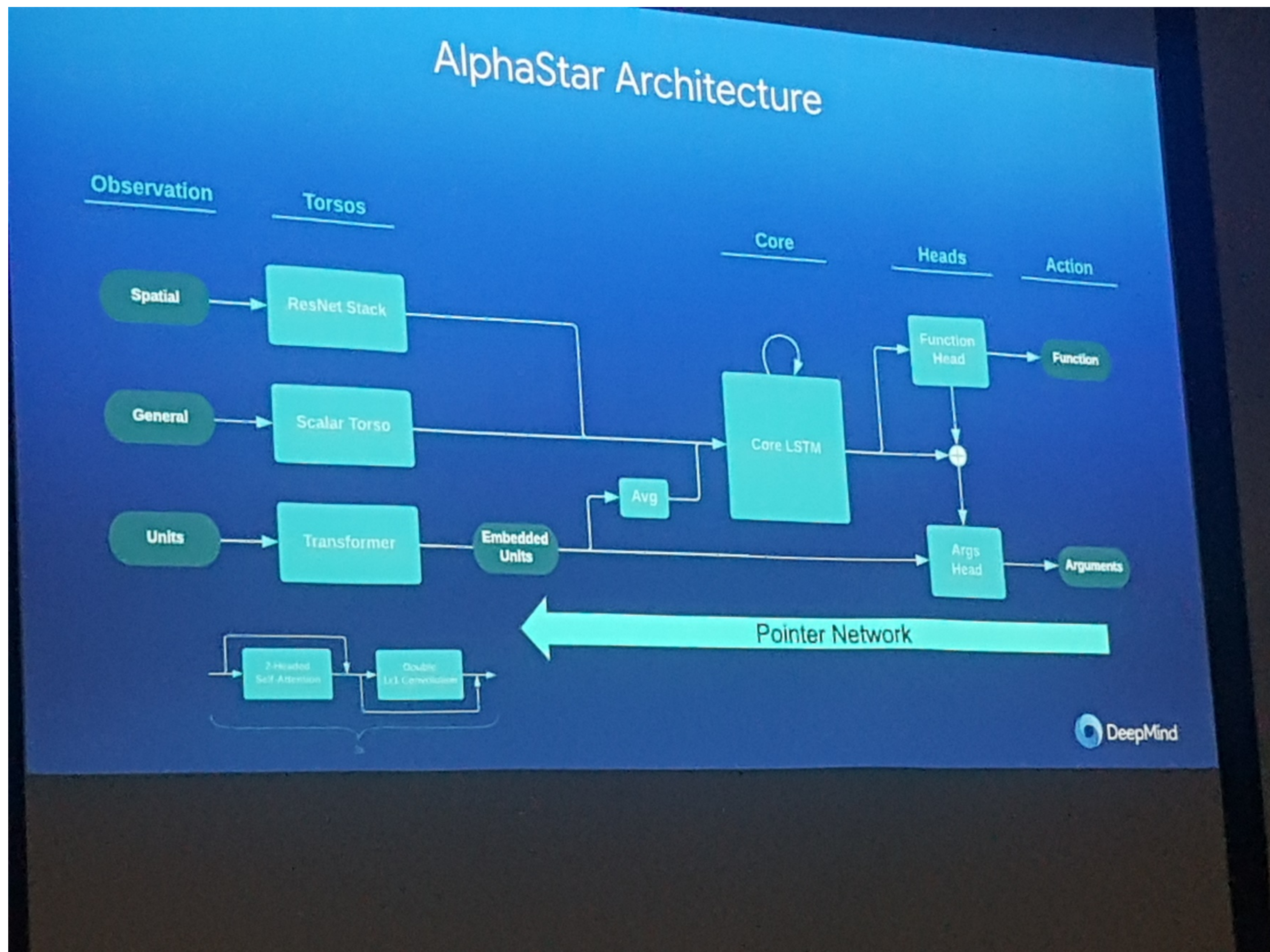
Example: AlphaStar

StarCraft2 Challenges

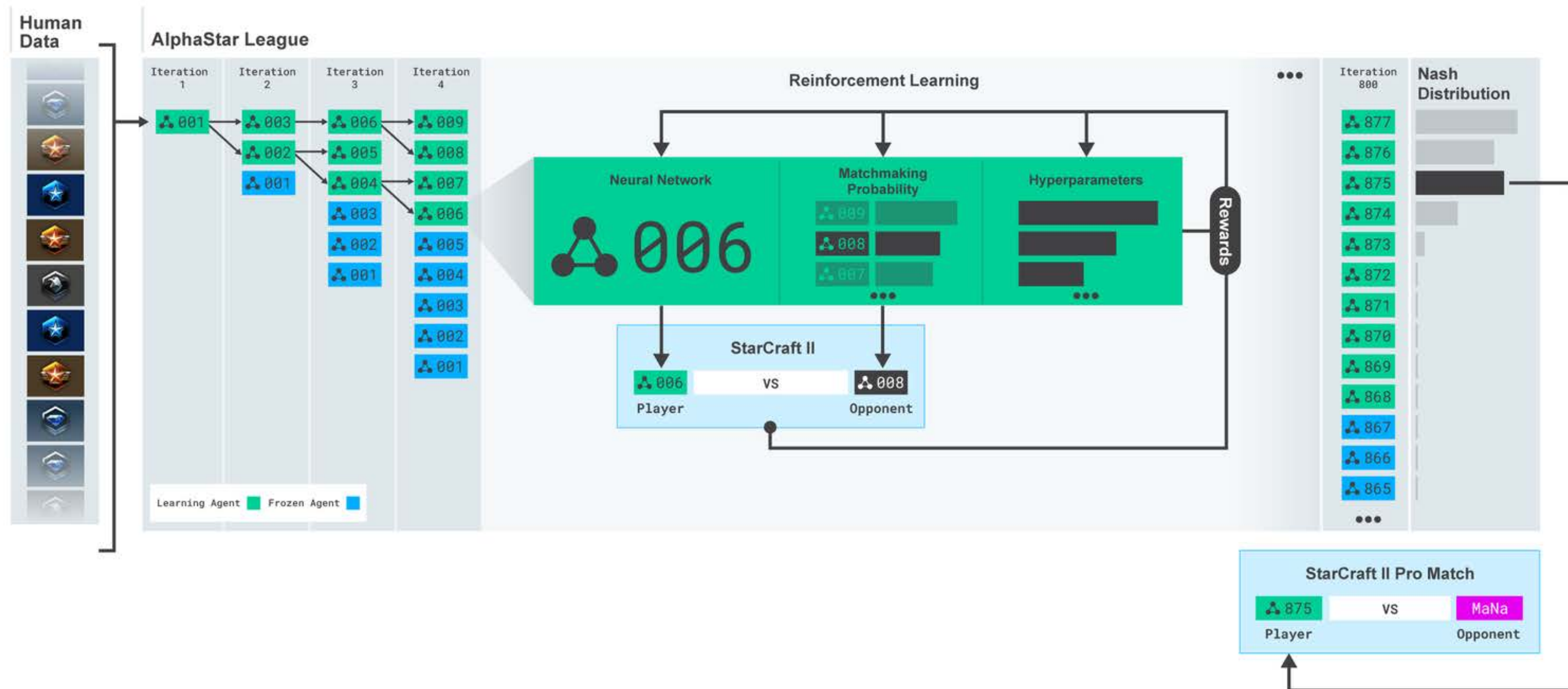
- game theory: there is no single strategy to win, strategic knowledge must constantly expanded
- Imperfect information: partially observable setting, with active choices to reduce uncertainty (scouting the map)
- Long term planning: the reward of winning has to spread to actions which led to success
- Real time: apm rates are important
- Large action space: Amount of possible moves is extremely high (units, builds,grouping,..)

"AlphaStar: Mastering the Real-Time Strategy Game StarCraft II" by Vinyals, Oriol and Babuschkin, Igor and Chung, Junyoung and Mathieu, Michael and Jaderberg, Max and Czarnecki, Wojciech M. and Dudzik, Andrew and Huang, Aja and Georgiev, Petko and Powell, Richard and Ewalds, Timo and Horgan, Dan and Kroiss, Manuel and Danihelka, Ivo and Agapiou, John and Oh, Junhyuk and Dalibard, Valentin and Choi, David and Sifre, Laurent and Sulsky, Yury and Vezhnevets, Sasha and Molloy, James and Cai, Trevor and Budden, David and Paine, Tom and Gulcehre, Caglar and Wang, Ziyu and Pfaff, Tobias and Pohlen, Toby and Wu, Yuhuai and Yogatama, Dani and Cohen, Julia and McKinney, Katrina and Smith, Oliver and Schaul, Tom and Lillicrap, Timothy and Apps, Chris and Kavukcuoglu, Koray and Hassabis, Demis and Silver, David https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ 2019

Architecture of AlphaStar



AlphaStar Training: AlphaStar League



- League of competitive agents + weaker or even trivial ones => be good against any opponent
- the 5 agents where selected and randomly chosen as opponent as opponents for the matches against TLO and MaNa from Team Liquid

Current Challenges

- Multi-Agent training: learn how to coordinate teams
  - What can and should be shared between agents?
  - How does the team goal reflect to the agents actions?
- Required resources:
  - Currently even running one agent is extremely resource intensive
  - How can these development be scaled to serve in MMOs?
- Challenges to game providers
  - Where does a more clever computer opponent make sense?
  - Where is selecting a human opponent easier?

Learning Goals

- agents and environments for sequential planning
- deterministic search
- building decision graph for routing in open environments
- Markov Decision Processes
- Policy and Value Iterations
- Model-free approaches and Q-Learning
- Function Approximation
- Antagonistic Search
- MiniMax Search and Alpha-Beta Pruning
- Monte Carlo Tree Search with UCT

Literature

- Nathan R. Sturtevant: **Memory-Efficient Abstractions for Pathfinding** In Artificial Intelligence and Interactive Digital Entertainment, Conference (AIIDE), 2007.
- Lecture notes D. Silver: Introduction to Reinforcement Learning (http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html)
- S. Russel, P. Norvig: Artificial Intelligence: A modern Approach, Pearson, 3rd edition, 2016
- Levente Kocsis and Csaba Szepesvári: Bandit based monte-carlo planning. In Proceedings of the 17th European conference on Machine Learning (ECML'06), 282-293, 2006
- V. Mnih, K. Kavokcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller: Playing Atari with Deep Reinforcement Learning, NIPS-DLW 2013.